OSProj7 Contiguous Memory Allocation

by 潘禧辰，518021910497

Menu

Abstract

- 编写了一个Contiguous Memory Allocation simulator，完成了Contiguous Memory Allocation的运行模拟
- 要求支持`RQ`、`RL`、`C`、`STAT`、`X`五种指令，分别完成request，release，compact，status report，exit的功能

Environment

- Ubuntu 18.04
- Linux 5.3.0-42-generic
- VMware Workstation Rro 15.5.0 build-14665864

Quick Start

编译

Contiguous Memory Allocation是用户态代码，直接使用如下gcc命令进行编译。

```
gcc allocator.c -o allocator
```

测试代码

```
./allocator 20000
allocator> RQ P0 5000 F
allocator> RQ P1 5000 F
allocator> RQ P2 5000 F
allocator> RQ P3 5000 F
allocator> STAT
allocator> RL P1
allocator> RL P3
allocator> STAT
allocator> RQ P4 2000 F
allocator> RQ P5 4000 B
allocator> RQ P6 1000 W
allocator> STAT
allocator> C
allocator> STAT
allocator> X
```

Implementation & Result

数据结构

定义了如下数据结构：

- `hole`结构体，定义每个`memory`中的`hole`
  - `use`表示是否被进程占用
  - `start`表示起始内存地址
  - `end`表示结束内存地址
  - `size`表示占用内存空间
  - `process`表示占用该`hole`的进程名字
- `memory`数组，由`hole`组成，表示内存
- `length`用于存储`memory`中`hole`的个数
- `cmd`数组，用于读取第一个参数`argv[1]`

```
typedef struct {
    bool use;
    int start;
    int end;
    int size;
    char process[10];
} hole;
hole memory[MAX_HOLE_NUM];
int length = 0;
char cmd[5];
```

main

定义了一些变量：

- `size`表示内存空间的总大小
- `process`表示`PQ`和`PL`的进程名
- `request_size`表示`PQ`需要分配的内存空间大小
- `strategy`表示`PQ`需要使用的分配方法

先完成了初始化和总内存空间的读取，再循环读取指令遇到`X`指令退出循环结束程序，遇到`RQ`指令调用`request`函数请求资源，遇到`RL`指令调用`release`释放资源，遇到`C`指令调用`compact`进行compact操作，遇到`STAT`指令调用`report`打印分配的内存和未使用的内存地址。

```
int main(int argc, char *argv[]) {
    assert(argc == 2);
    int size = atoi(argv[1]);
    char process[10];
    int request_size;
    char strategy;

    memory[0].use = false;
    memory[0].start = 0;
    memory[0].end = size - 1;
    memory[0].size = size;
    length++;

    for (int i = 1; i < MAX_HOLE_NUM; i++)
        memory[i].use = false;

    while (true) {
        printf("allocator> ");
        scanf("%s", cmd);
        if (strcmp(cmd, "X") == 0)
            break;
        else if (strcmp(cmd, "RQ") == 0) {
            scanf(" %s %d %ch", process, &request_size, &strategy);
            request(process, request_size, strategy);
        }
        else if (strcmp(cmd, "RL") == 0) {
            scanf(" %s", process);
            release(process);
        }
        else if (strcmp(cmd, "C") == 0)
            compact();
        else if (strcmp(cmd, "STAT") == 0)
            report();
        else
            printf("error input\n");
    }
    return 0;
}
```

request

先根据`strategy`判断内存分配方式，针对每种方式找到合适的进行分配的`hole`的地址`index`，如果`hole`的大小正好是进程需要的大小，那么直接变更`hole`的状态`use`和进程名`process`即可；如果不是正好，那么需要将`index`及其之后的所有`hole`后移一个，将进程分配到`memory[index]`中，将剩余空余空间分配到`memory[index+1]`中，并变更这两个`hole`的属性

```
bool request(char process[], int request_size, char strategy) {
    int index = 0;
    int i;
    bool flag = false;
    if (strategy == 'F') {
```

```
        for (i = 0; i < length; i++)
            if (!memory[i].use && memory[i].size >= request_size) {
                flag = true;
                break;
            }
        index = i;
    }
    else if (strategy == 'B') {
        int min = INT_MAX;
        for (i = 0; i < length; i++)
            if (!memory[i].use && memory[i].size >= request_size &&
memory[i].size < min) {
                flag = true;
                index = i;
                min = memory[i].size;
            }
    }
    else if (strategy == 'W') {
        int max = 0;
        for (i = 0; i < length; i++)
            if (!memory[i].use && memory[i].size >= request_size &&
memory[i].size > max) {
                flag = true;
                index = i;
                max = memory[i].size;
            }
    }
    else {
        printf("error strategy\n");
        return false;
    }

    if (!flag) {
        printf("no space to allocate\n");
        return false;
    }

    if (memory[index].size == request_size) {
        strcpy(memory[index].process, process);
        memory[index].use = true;
        return true;
    }

    for (i = length - 1; i >= index; i--) {
        memory[i + 1].use = memory[i].use;
        memory[i + 1].start = memory[i].start;
        memory[i + 1].end = memory[i].end;
        memory[i + 1].size = memory[i].size;
        strcpy(memory[i + 1].process, memory[i].process);
    }
    length++;
    memory[index].use = true;
    memory[index].end = memory[index].start + request_size - 1;
    memory[index].size = request_size;
    strcpy(memory[index].process, process);
    memory[index + 1].start = memory[index].end + 1;
    memory[index + 1].size = memory[index + 1].end - memory[index + 1].start +
1;
```

```
    return true;
}
```

release

先根据进程名`process`寻找到需要`RL`的进程所处在的`hole`，将其下标存入`index`，将`memory[index]`的状态`use`进行改变即可。

但是由于此时可能有两个相邻的空闲`hole`，需要检查`memory[index]`的左右两边，看看其能否与左右的`hole`合并

```
void release(char process[]) {
    int index;
    int i;
    for (i = 0; i < length; i++)
        if (memory[i].use && strcmp(process, memory[i].process) == 0)
            break;
    if (i == length) {
        printf("process not found\n");
        return;
    }
    index = i;

    memory[index].use = false;
    if (index < length - 1 && !memory[index + 1].use) {
        memory[index].end = memory[index + 1].end;
        memory[index].size += memory[index + 1].size;
        for (i = index + 1; i < length; i++) {
            memory[i].use = memory[i + 1].use;
            memory[i].start = memory[i + 1].start;
            memory[i].end = memory[i + 1].end;
            memory[i].size = memory[i + 1].size;
            strcpy(memory[i].process, memory[i + 1].process);
        }
        length--;
    }
    if (index >= 1 && !memory[index - 1].use) {
        memory[index - 1].end = memory[index].end;
        memory[index - 1].size += memory[index].size;
        for (i = index; i < length; i++) {
            memory[i].start = memory[i + 1].start;
            memory[i].end = memory[i + 1].end;
            memory[i].size = memory[i + 1].size;
            strcpy(memory[i].process, memory[i + 1].process);
            memory[i].use = memory[i + 1].use;
        }
        length--;
    }
}
```

compact

定义了一些变量：

- `used_hole_index`表示正在被进程占用的`hole`的地址
- `cnt`表示正在被进程占用的`hole`的数量

- `unused`表示未被进程占用的空闲`hole`的总大小

先遍历`memory`统计上述三个变量，从`memory`头部开始对`used_hole_index`按顺序进行分配，第`cnt`个`hole`用于存放空闲内存，将第`cnt`个之后的`hole`状态`use`全部置为`false`，并更新`length`大小为`cnt+1`

```
void compact() {
    int used_hole_index[MAX_HOLE_NUM];
    int cnt = 0;
    int unused = 0;
    for (int i = 0; i < length; i++) {
        if (memory[i].use) {
            used_hole_index[cnt] = i;
            cnt++;
        }
        else
            unused += memory[i].size;
    }

    if (cnt == 0) return;

    for (int i = 0; i < cnt; i++) {
        memory[i].use = true;
        memory[i].start = i > 0 ? memory[i - 1].end + 1 : 0;
        memory[i].end =  memory[i].start + memory[used_hole_index[i]].size - 1;
        memory[i].size = memory[used_hole_index[i]].size;
        strcpy(memory[i].process, memory[used_hole_index[i]].process);
    }
    memory[cnt].use = false;
    memory[cnt].start = memory[cnt - 1].end + 1;
    memory[cnt].end = memory[cnt].start + unused - 1;
    memory[cnt].size = unused;
    for (int i = cnt + 1; i < MAX_HOLE_NUM; i++)
        memory[i].use = false;
    length = cnt + 1;
}
```

report

循环按照格式打印分配的内存和未使用的内存地址。

```
void report() {
    for (int i = 0; i < length; i++) {
        if (memory[i].use)
            printf("Addresses [%d:%d] Process %s\n", memory[i].start,
memory[i].end, memory[i].process);
        else
            printf("Addresses [%d:%d] Unused\n", memory[i].start,
memory[i].end);
    }
}
```

完整代码

将上述几个部分组合得到：

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <limits.h>
#include <assert.h>

#define MAX_HOLE_NUM 50

typedef struct {
    bool use;
    int start;
    int end;
    int size;
    char process[10];
} hole;

bool request(char process[], int request_size, char strategy);

void release(char process[]);

void compact();

void report();

hole memory[MAX_HOLE_NUM];
int length = 0;
char cmd[5];

int main(int argc, char *argv[]) {
    assert(argc == 2);
    int size = atoi(argv[1]);
    char process[10];
    int request_size;
    char strategy;

    memory[0].use = false;
    memory[0].start = 0;
    memory[0].end = size - 1;
    memory[0].size = size;
    length++;

    for (int i = 1; i < MAX_HOLE_NUM; i++)
        memory[i].use = false;

    while (true) {
        printf("allocator> ");
        scanf("%s", cmd);
        if (strcmp(cmd, "X") == 0)
            break;
        else if (strcmp(cmd, "RQ") == 0) {
            scanf(" %s %d %ch", process, &request_size, &strategy);
            request(process, request_size, strategy);
        }
        else if (strcmp(cmd, "RL") == 0) {
            scanf(" %s", process);
            release(process);
```

```
        }
        else if (strcmp(cmd, "C") == 0)
            compact();
        else if (strcmp(cmd, "STAT") == 0)
            report();
        else
            printf("error input\n");
    }
    return 0;
}

bool request(char process[], int request_size, char strategy) {
    int index = 0;
    int i;
    bool flag = false;
    if (strategy == 'F') {
        for (i = 0; i < length; i++)
            if (!memory[i].use && memory[i].size >= request_size) {
                flag = true;
                break;
            }
        index = i;
    }
    else if (strategy == 'B') {
        int min = INT_MAX;
        for (i = 0; i < length; i++)
            if (!memory[i].use && memory[i].size >= request_size &&
memory[i].size < min) {
                flag = true;
                index = i;
                min = memory[i].size;
            }
    }
    else if (strategy == 'W') {
        int max = 0;
        for (i = 0; i < length; i++)
            if (!memory[i].use && memory[i].size >= request_size &&
memory[i].size > max) {
                flag = true;
                index = i;
                max = memory[i].size;
            }
    }
    else {
        printf("error strategy\n");
        return false;
    }

    if (!flag) {
        printf("no space to allocate\n");
        return false;
    }

    if (memory[index].size == request_size) {
        strcpy(memory[index].process, process);
        memory[index].use = true;
        return true;
    }
```

```
    for (i = length - 1; i >= index; i--) {
        memory[i + 1].use = memory[i].use;
        memory[i + 1].start = memory[i].start;
        memory[i + 1].end = memory[i].end;
        memory[i + 1].size = memory[i].size;
        strcpy(memory[i + 1].process, memory[i].process);
    }
    length++;
    memory[index].use = true;
    memory[index].end = memory[index].start + request_size - 1;
    memory[index].size = request_size;
    strcpy(memory[index].process, process);
    memory[index + 1].start = memory[index].end + 1;
    memory[index + 1].size = memory[index + 1].end - memory[index + 1].start +
1;
    return true;
}

void release(char process[]) {
    int index;
    int i;
    for (i = 0; i < length; i++)
        if (memory[i].use && strcmp(process, memory[i].process) == 0)
            break;
    if (i == length) {
        printf("process not found\n");
        return;
    }
    index = i;

    memory[index].use = false;
    if (index < length - 1 && !memory[index + 1].use) {
        memory[index].end = memory[index + 1].end;
        memory[index].size += memory[index + 1].size;
        for (i = index + 1; i < length; i++) {
            memory[i].use = memory[i + 1].use;
            memory[i].start = memory[i + 1].start;
            memory[i].end = memory[i + 1].end;
            memory[i].size = memory[i + 1].size;
            strcpy(memory[i].process, memory[i + 1].process);
        }
        length--;
    }
    if (index >= 1 && !memory[index - 1].use) {
        memory[index - 1].end = memory[index].end;
        memory[index - 1].size += memory[index].size;
        for (i = index; i < length; i++) {
            memory[i].start = memory[i + 1].start;
            memory[i].end = memory[i + 1].end;
            memory[i].size = memory[i + 1].size;
            strcpy(memory[i].process, memory[i + 1].process);
            memory[i].use = memory[i + 1].use;
        }
        length--;
    }
}
```

```
void compact() {
    int used_hole_index[MAX_HOLE_NUM];
    int cnt = 0;
    int unused = 0;
    for (int i = 0; i < length; i++) {
        if (memory[i].use) {
            used_hole_index[cnt] = i;
            cnt++;
        }
        else
            unused += memory[i].size;
    }

    if (cnt == 0) return;

    for (int i = 0; i < cnt; i++) {
        memory[i].use = true;
        memory[i].start = i > 0 ? memory[i - 1].end + 1 : 0;
        memory[i].end =  memory[i].start + memory[used_hole_index[i]].size - 1;
        memory[i].size = memory[used_hole_index[i]].size;
        strcpy(memory[i].process, memory[used_hole_index[i]].process);
    }
    memory[cnt].use = false;
    memory[cnt].start = memory[cnt - 1].end + 1;
    memory[cnt].end = memory[cnt].start + unused - 1;
    memory[cnt].size = unused;
    for (int i = cnt + 1; i < MAX_HOLE_NUM; i++)
        memory[i].use = false;
    length = cnt + 1;
}

void report() {
    for (int i = 0; i < length; i++) {
        if (memory[i].use)
            printf("Addresses [%d:%d] Process %s\n", memory[i].start,
memory[i].end, memory[i].process);
        else
            printf("Addresses [%d:%d] Unused\n", memory[i].start,
memory[i].end);
    }
}
```

结果

```
pan@pan-virtual-machine:~/桌面/osproj/7$ gcc allocator.c -o allocator
pan@pan-virtual-machine:~/桌面/osproj/7$ ./allocator 20000
allocator> RQ P0 5000 F
allocator> RQ P1 5000 F
allocator> RQ P2 5000 F
allocator> RQ P3 5000 F
allocator> STAT
Addresses [0:4999] Process P0
Addresses [5000:9999] Process P1
Addresses [10000:14999] Process P2
Addresses [15000:19999] Process P3
allocator> RL P1
allocator> RL P3
allocator> STAT
Addresses [0:4999] Process P0
Addresses [5000:9999] Unused
Addresses [10000:14999] Process P2
Addresses [15000:19999] Unused
allocator> RQ P4 2000 F
allocator> RQ P5 4000 B
allocator> RQ P6 1000 W
allocator> STAT
Addresses [0:4999] Process P0
Addresses [5000:6999] Process P4
Addresses [7000:7999] Process P6
Addresses [8000:9999] Unused
Addresses [10000:14999] Process P2
Addresses [15000:18999] Process P5
Addresses [19000:19999] Unused
allocator> C
allocator> STAT
Addresses [0:4999] Process P0
Addresses [5000:6999] Process P4
Addresses [7000:7999] Process P6
Addresses [8000:12999] Process P2
Addresses [13000:16999] Process P5
Addresses [17000:19999] Unused
allocator> X
pan@pan-virtual-machine:~/桌面/osproj/7$
```

Difficulties

- 整体比较简单，重点和难点在于理解`memory`数据结构应当如何组织

Reference

- Operating System Concept 10^{th} edition
- Source code for the 10th edition of Operating System Concepts https://github.com/greggagne/osc10e